Expressions

- **expression** - a combination of values, function calls and operations that, when evaluated, produces an output value
- may or may not have side effects

Expression Evaluation Order

```
int c = 1, d = 2;

int f() {
  ++c;
  return 0;
}

int main() {
  cout << c - d + f() ;  // produces either -1 or 0
}
```

- C++ doesn't specify what order the terms of an arithmetic expression must be evaluated!
- Nor does it specify what order parameters must be evaluated!
- Some languages mandate a left-to-right eval order (C#, Java, JavaScript, Swift), while others don't (C++, Rust, OCaml).
- in general, if your expressions are calling functions, they shouldn't have side effects that could impact the execution of the other functions called in the expression
    - Otherwise you'll get unpredictable behavior depending on the compiler and potentially each time you compile the code

> *Why avoid specifiying one explicit evaluation order? Not specifying ordering gives the compiler latitude to optimize the code*

Associativity

- most languages use left-to-right associativity when evaluating mathematical expressions
    - some notable exceptions: exponentiation and assignment are evaulated right to left

```
  4^3^2 // equivalent to
  4^(3^2)

  a = b = c = 5; // equivalent to:
  a  = (b = (c = 5));
```

Short Circuiting

- In any boolean expression with AND / OR, most languages will evaluate its sub-expressions from left-to-right
- The moment the language finds a condition that satisfies or invalidates the boolean expression, it will not evaluate the rest

- This can dramatically speed up your code!

```
if(is_cute() || is_smart() || is_rich()) {
  date_person();
}
// equivalent to
if (is_cute())
  date_person();
else if (is_smart())
  date_person();
else if (is_rich())
  date_person();
```

```
if (good_salary() && fun_work() && good_culture()) {
  take_job();
}
// equivalent to
if (good_salary()) {
  if (fun_work()) {
    if (good_culture()) {
      take_job();
    }
  }
}
```

Control Flow

Unconditional Branching

goto

- can assign **labels** to locations in your program and jump to them
- frowned upon

Why goto is bad:

- Improper use of GOTO statements can lead to "spaghetti" code with difficult to understand semantics.
- code using structure programming, with clear nested blocks and loops or switches, is easy to follow; its flow of control is very predictable. It's therefore easier to ensure that invariants are respected.

It is about scope. One of the main pillars of good programming practice is keeping your scope tight. You need tight scope because you lack the mental ability to oversee and understand more than just a few logical steps. So you create small blocks (each of which becomes "one thing") and then build with those to create bigger blocks which become one thing, and so on. This keeps things manageable and comprehensible.

A goto effectively inflates the scope of your logic to the entire program. This is sure to defeat your brain for any but the smallest programs spanning only a few lines.

break and continue

- `continue` statement jumps to the top of the innermost loop and advances to the next loop iteration.

    - **labeled continue** immediately breaks out of all inner loops and jumps to the labelled loop for the next iteration
- `break` statement terminates the innermost loop and execution jumps to the statement following it.
    - **labeled break** will break out of the labeled loop

```
// JavaScript
outer_loop: for (i = 0; i < 100000; i++) {
  console.log("outer: " + i);
  for (j = 0; j < 100000; j++) {
    console.log("inner: " + j);
    if (j == 2) break outer_loop;
  }
}

console.log("done!");

/*
outer: 0
inner: 0
inner: 1
inner: 2
done!
*/
```

Conditional Branching

- if statements
- conditional ternary operator
- **null coalescing operator**: if A is not null, return A, else B
    - Kotlin: (elvis operator) `?:` - `A ?: B`
    - Javascript, C# : `??`
- **safe navigation operator**: accesses a field unless the value is null, then null is returned
    - javascript: `MaybeNull?.field`

multiway selection

- switch statement in C++
    - can have multiple cases fall through to same code
    - explicit `break`
    - all cases must be constant char/int/enum values
- semantics in varous other languages:
    - require default case
    - don't need explicit `break`
    - can have non-scalar values for labels
    - pattern matching

Loops

Iteration/Looping

Counter-controlled: like we would in a for-loop in C++ or using a range in Python:

```
    for i in range(1,10):
        do_something()
```

Condition-controlled: use a boolean condition to decide how to long to loop - this is almost always a while() or do-while() loop.

```
    while (condition)
       do_something()
```

Collection-controlled: We're enumerating the items in a collection

```
   for (x in container)
        do_something(x)
```

- using an iterator (of which there are two types)
- or using first-class functions

Iterators:

```
for n in student_names:
 print(n)

for i in range(0,10):
  print(i)
```

First-class functions:

```
fruits.forEach
    { f -> println("$f") }
```

Iterator-based loops

- **iterable object** - an object that holds, has access to, or can generate a sequence of values
    - container objects (arrays, lists, tuples, sets, dicts)
    - I/O objects (disk file, network stream)
    - **generator objects** - objects that can generate a sequence
- **iterator** - an object that enables enumeration of an iterable object
- can enumerate
    - values in a container
    - contents of external sources like data files
    - **abstract sequence** - one where the values in the sequence aren't stored anywhere, but generated as they're retrieved via the iterator
- allows access the values of a sequence without exposing underlying implementation
- to get an iterator, you request one from an iterable object
- Iterators used with containers and external sources are typically distinct objects from the objects that contain the sequences they refer to

```
// Swift: abstract sequence
var range = 1...1000000
var it = range.makeIterator()  // get an iterator to the first # of the abstract seq.
```

```
// C++
vector<string> fruits = {"apple", "pear", ... };

auto it = fruits.begin();  // get an iterator into a C++ vector
```

Iterator Interface

Iterators in most languages have an interface that look like one of these (C++ has a notably different approach)

Interface: hasNext() and next()

- used with containers and external sources like data files
- `iter.hasNext()` : ask the iterator (NOT the iterable object) whether it refers to a valid value that can be retrieved
- `iter.next()` : ask the iterator (NOT the iterable object) to get the value that it points to, and advance to the next item

```
// Kotlin: iterator into container
val numbers = listOf(10,20,30,42)

val it = numbers.iterator()
while (it.hasNext()) {
  val v = it.next();
  println(v)
}
```

Interface: only next()

- used with abstract sequences, like ranges: range(1,10)
- `iter.next()` : The iterator (NOT the iterable object) generates and returns the next value in the sequence, or indicates the sequence is over via a return code or by throwing an exception

```
// Swift: abstract sequence
var range = 1...1000000

var it = range.makeIterator()
while true {
  var v = it.next()  // in Swift, the next() method returns either nil or a valid
value
  if (v == nil) { break }
  print(v)
}
```

C++ has unique iterators

- C++ models its iterators after C++ pointers, which can be incremented, decremented, and dereferenced
- They're a notable exception to iterators in most languages!

```
// C++ has unusual iterator syntax!
vector<string> fruits = {"apple", "pear", ... };

for (auto it = fruits.begin(); it != fruits.end(); ++it)
  cout << *it;
```

Looping under the hood

- when looping over an iterable, the language secretly uses an iterator to move through the items! Syntactic Sugar!

```
// Kotlin: iterate over list of #s
val numbers = listOf(10,20,30,42)

for (v in numbers) {
  println(v)
}

// what's actually happening:

val it = numbers.iterator()
while (it.hasNext()) {
  val v = it.next()
  println(v)
}
```

```
// Swift: iterate over range
for v in 1...1000000 {
  print(v)
}

// what's actually happening:

var range = 1...1000000

var it = range.makeIterator()
while true {
  var v = it.next()
  if (v == nil) { break }
  print(v)
}
```

Iterator implementation

Two primary approaches:

- Traditional Classes: An iterator is an object, implemented using regular OOP classes.
- "True Iterators" aka **Generators**: An iterator is implemented by using a special type of function called a generator.

Iterators: traditional classes

- define an iterable class (e.g., like a vector class) which we can iterate over

- in Python, to make a class iterable, define an `__iter__()` method that creates and returns an iterator object
    - `ListsOfStrings` is an *iterable* class - NOT an iterator

```
class ListOfStrings:
  def __init__(self):
    self.array = []

  def add(self,val):
    self.array.append(val)

  def __iter__(self):
    it = OurIterator(self.array)  # calling __iter__ returns a new OurIterator object
    return it
```

Here's the iterator class, notice that it's uniqely tied to `ListOfStrings`

```
class OurIterator:
  def __init__(self, arr):
    self.arr = arr
    self.pos = 0

  def __next__(self):
    if self.pos < len(self.arr):
      val = self.arr[self.pos]
      self.pos += 1
      return val
    else:
      raise StopIteration
```

- iterator class must have a method called `__next__()` that gets the value of the item pointed to by the iterator, advances the iterator, and returns the value.
- if the iterator has run out of items to iterate over, the iterator (in Python) will throw an exception to tell the user of the iterator that there are no more items left

```
nerds = ListOfStrings()
nerds.add("Carey")
nerds.add("David")
nerds.add("Paul")

for n in nerds:  # this is now possible
  print(n)
```

"True Iterators" aka Generators

- "true iterator" - an iterator that is implemented by using a special type of function called a generator
- **generator** - essentially a closure that can be paused and resumed over time

```
# pseudocode
def foo(n):
```

```python
  for i in range (1, n):
    print(f'i is {i}')
    yield
    print('woot!')

def main():
  p = foo(4)  # Line A
  print('CS')
  next(p)       # Line B
  print('is')
  next(p)      # Line C
  print('cool!')
  next(p)

  # CS
  # i is 1
  # is
  # woot!
  # i is 2
  # cool!
  # woot!
  # i is 3
```

Line A:

- calling a generator (implicitly denoted with `yield` statement) does NOT run the code
- it creates a special closure, which has an extra piece of data - an instruction pointer pointing to first line in function
- function starts in a suspended state! It's not running, we just created the closure

Line B:

- start running the generator until it hits `yield`
- the instruction pointer is saved in the closure so we know where to continue running, and all changes made to the local variables in the closure are retained
- the generator returns and the next line of the main function starts running

Line C:

- resume the generator, and it continues running where it left off last time

`yield` can also return a value

```python
def bar(a, b):
  while a > b:
    yield a  # Line A
    a -= 1

f = bar(3, 0)
print('Eat prunes!')
t = next(f)  # Line B
print(t)
```

```
t = next(f)
print(t)
t = next(f)
print(t)
print(f'Explosive diarrhea!')

# Eat prunes!
# 3
# 2
# 1
# Explosive diarrhea!
```

- This is a more idiomatic use of a generator – to "generate" a sequence of values that can be retrieved one at a time – the sequence might be finite, or infinite!
- Since a generator is an iterable object, and produces an iterator, it can be used like any other iterable object in loops!

```
def our_range(a, b):
  while a > b:
    yield a
    a -= 1

print('Eat prunes!')
for t in our_range(3,0):
  print(t)
print(f'Explosive diarrhea!')
```

Class-based Iterators vs Generators

- Some things are easier with class-based iterators, some things are easier with generators
- But the two are isomorphic and functionally equivalent - anything we can do with one we can with the other

First-class function-based iteration

- the iterable object contains a `forEach()` method, which accepts a function that is run on each item in the iterable
- NOT using an iterator, handled internally by iterable
- like `map` but runs a function instead of returning an item

```
// Rust
(0..10).for_each(|elem| println!("{}", elem));
```

Examples

```
def take(n, gen):  # gen is created upon call: take(1, gen(3))
    return [next(gen) for _ in range(n)]
```

131 notes

Taylor Chen **tchen073@gmail.com**

AttachmentsDec 11, 2023, 4:54 PM (2 days ago)

to me

20 Attachments • Scanned by Gmail

Prolog and Logic Programming

Logic programming is a paradigm where we express programs as a set of facts (relationships which are held to be true), and a set of rules (i.e. if A is true, then B is true).

- A program/user then issues queries against this these facts and axioms:

- Logic programming is declarative – programs specify "what" they want to compute, not "how" to do so.

Example:

Facts:

- Martha is Andrea's parent
- Andrea is Carey's parent

Rules: If X is the parent of Q, and Q is the parent of Y, then X is the grandparent of Y

Queries:

- Is Martha the grandparent of Carey?
- Who is the grandparent of Carey?

Prolog Facts and Rules

Prolog programs are comprised of facts about the world, and rules that can be used to discover new facts.

Facts

fact - a predicate expression that declares either

- An attribute about some thing (aka an atom); you can think of this as being "always true"
  - `outgoing(ren)`
- A relationship between two or more atoms or numbers:
  - `parent(alice, bob)`
  - `age(ren, 80)`
  - `teaches(carey, course(cs,131))`

terms:

- an **atom** is a "thing"; (`ren` , `alice` , `bob` , 80)
- a **functor** is a static assertion (a relationship); (`outgoing` , `parent` , and `age`)
  - the order of arguments to functors is defined by the user, just stay consistent!
- atoms and functors *must be lowercase*

closed-world assumption (CWA) - only things that can be proven true by the program are true; *everything else* is false

Rules

A **rule** lets us define a new fact in terms of one or more existing facts or rules. Two parts:

- **head** - defines what the rule is trying to determine
- **body** - specifies the conditions (aka subgoals) that allow us to conclude the head is true

Rules can be defined with atoms, numbers, or *variables*

- **Variables** (like `X` and `Y` below) are like placeholders which Prolog will try to fill in as it tries to answer user queries
- *Variables must always be capitalized*

```
% Facts:
outgoing(ren).
silly(ren).
parent(alice, bob).
parent(bob, carol).

% Rules:
comedian(P) :- silly(P), outgoing(P).
grandparent(X, Y) :- parent(X,Q), parent(Q,Y).
```

- `:-` means "if"
- `,` in the body means AND

Recursive Rules

- Rules can also be recursive and can have multiple parts!
- Similar to recursion and pattern matching in Haskell, Prolog processes rules from the top to the bottom. So the base case(s) should always go first!

```
% Recursive rules:
ancestor(X, Z) :- parent(X,Z).
ancestor(X, Z) :- parent(X,Y), ancestor(Y,Z).
```

X is the ancestor of Z if: X is Z's parent OR if X is some person Y's parent AND Y is an ancestor of person Z

Negation in Rules

`not(something)` works as follows:

1. Prolog tries to prove `something` is true using all of the program's facts and rules
2. If `something` can't be proven as true, then `not(something)` is true

```
% Rules with negation:
serious(X) :- not(silly(X)).
```

For example, the query `serious(alice)` is true, since the closed-world assumption says that `silly(alice)` is false.

Prolog Queries

We can create queries to answer true/false questions:

- A query can match a simple fact...
- Or it can execute a rule.

```
?- grandparent(brenda, ned)
true
```

We can also create queries to fill-in-the-blanks:

- The query will find ALL possible matches.
- The query can specify multiple unknowns.
- And Prolog will find ALL consistent combinations of answers! (not just the first)

```
?- parent(alice , Who)
Who = bob
Who = brenda
?- grandparent(A, B)
A = alice, B = caitlin
A = brenda, B = ned
```

Resolution: How Prolog Answers Queries

Algorithm:

1. Add our query to a goal stack
2. Pop the top goal in our goal stack and match it with each item in our database, from top to bottom
3. If we find a match, extract the variable mappings, and create a new map with the old+new mappings
4. Create a new goal stack, copying over existing, unprocessed goals and adding new subgoals
5. Recursively repeat the process on the new goal stack and the new mappings

```
parent (nel, ted).
parent (ann, bob).
parent (ann, bri).
parent (bri, cas).
gparent( X ,  Y ) :- parent(X,Q), parent(Q,Y).
```

Query:

```
gparent(ann, W)
```

Initially, goal stack is `gparent(ann,W)`, and mappings are `{}`. Then, after matching the last rule, our goal stack becomes:

```
# goal stack:
parent(X,Q)
parent(Q,Y)

# mappings:
{X->ann,W<->Y}
```

Then, pop the top rule off our goal stack. We have a potential match with `parent(nel, ted)`

```
# goal stack:
parent(Q,Y)

# mappings:
{X->ann,W<->Y, Q->bob}
```

Pop the top rule off the stack again. There's no rule that matches `parent(bob,Y)`, which means the set of mappings we discovered were not valid.

So back-track one level up and continue searching for alternative mappings of `Q`

If our goal stack is empty, we output variables from our mapping that were explicitly queried; in this case, our final mapping is: `{ X ->ann, W <-> Y <-> cas, Q ->bri }`

So we return `cas`, since `W=cas`.

Unification

- Unification is used within the broader Resolution algorithm
- In Resolution, Prolog repeatedly compares the current goal with each fact/rule to see if they are a match
- If a goal and a fact/rule match, map between variables and atoms (e.g., X->bob)

Unification is the process of:

- comparing a single goal with a single fact/rule, given the current set of mappings
- determining if the goal and the fact/rule match
- extracting all new mappings between variables and atoms on either side

```
# Unification pseudocode:
def unify(goal, fact_or_rule, existing_mappings):
   if the goal with the existing mappings matches the current fact/rule:
      mappings = extract variable mappings between goal and fact/rule
      return (True, mappings)     # We did unify! Return discovered mappings
   otherwise:
      return (False, {})       # We couldn't unify! So no mappings found
```

- The unification algorithm runs on one goal at a time.
- Unification runs on a single fact or rule at a time.

Matching a Goal and a Fact/Rule

How do we determine if a goal with the current mappings matches a specific fact/rule?

1. apply all current mappings to the goal
2. treat both the mapped goal and head of the fact/rule as trees
3. Compare each node of the goal tree with the corresponding node in the fact/rule tree

Use the following comparison rules:

- If both nodes are functors, then make sure the functors are the same and have the same number of children.
- If both nodes hold atoms, make sure the atoms are the same.
- If a goal node holds an unmapped variable it will match ANY item in the corresponding node of the fact/rule.
- If a fact/rule node holds an unmapped variable it will match ANY item in the corresponding node of the goal.

Unification: Do They Match

Example:

Assuming we have the mapping `{What->ucla}`, and we are trying to match (1) `likes(gene, What)` with (2) `likes(Y,X)`. These *DO MATCH*.

- we replace `What` with `ucla`, so the children of the `likes` node for (1) becomes `gene` and `ucla`.
- This matches the children nodes for (2), since they both hold unmapped variables [`Y` and `X` respectively]

Unification: Extracting Variable Mappings

Once we know that a goal matches a fact/rule, we need to extract the variable mappings.

- Iterate through each corresponding pair of nodes in both trees.
- when you find an unmapped Variable in either tree that maps to an atom/number in the other tree, create a mapping between the variable and the atom.
- when you find an unmapped Variable in either tree that maps to an unmapped Variable in the other tree, create a bidirectional mapping between the variables.

Unification in Resolution

A simplified version of the Resolution algorithm, showing where Unification fits

```
def resolution(database, goals, cur_mappings):
  if there are no goals left:
    tell the user we found a solution and output our discovered mappings!
    return
  for each fact/rule z in the database:
    success, new_mappings = unify(goals[0], fact_or_rule, cur_mappings)
    if success:
      tmp_mappings = cur_mappings + new_mappings
      tmp_goals = sub_goals(z) + goals[1:]
      resolution(database, tmp_goals, tmp_mappings)    # recursion
  # if we get here, we didn't find a match... BACKTRACK and keep trying!
```

The Resolution algorithm is initially called with the user's query as its only goal, and with no initial mappings.

Prolog Lists

- Lists can contain numbers, atoms, or other lists, don't have to have all the same type
- Prolog uses a combination of pattern matching (like Haskell) and unification to process lists.
- List processing is also done with facts and rules, just like other inference tasks.

Firstly, here's a Prolog fact with **variables** inside of atoms.

```
is_the_same(X,X)

is_the_same(lit,lit) --> returns true
is_the_same(ucla,usc) --> returns false
```

- Prolog is unifying from left-to-right, mapping `lit` to `X` and `ucla` to `X`

Example

```
is_head_item(X,[X | XS]).

is_head_item(lit, [lit, dank, snack]) --> returns true
is_head_item(drip, [lit,dank,snack]) --> returns false
```

- This [X | XS] syntax is Prolog's equivalent of pattern matching, like (x:xs) in Haskell
- Like above Prolog unifies from left-to-right and maps each variable
- Once it extracts a mapping, it only "unifies" the query if later uses of the mapping are consistent with the first.

```
is_second_item(Y, [X, Y | XS]).

is_head_item(dank, [lit, dank, snack]) --> returns true
is_head_item(lit, [lit,dank,snack]) --> returns false
```

`[X, Y | XS]` equivalent of `(x:y:xs)` in Haskell.

Check if list contains value:

```
is_member(X,[X|Tail]).
is_member(Y,[Head|Tail]) :- is_member(Y,Tail).
```

If we query `is_member(dank,[lit,dank,snack])`:

- we first try and match the first fact, but it's false
- so we match on the second rule, which turns out to unify, so we return True!

Deleting an atom from a list

general form: `delete(ItemToDelete, ListToDeleteFrom, ResultingList)`

- notice that the "output" is also an argument!

Query: `?- delete(carey, [paul, carey, david], X) --> X = [paul, david]`

```
delete(Item, [], [])
delete(Item, [Item | Tail], Tail).
delete(Item_, [Head_ | Tail_], [Head_ | FinalTail])  :-  delete(Item_, Tail_,
FinalTail).
```

- The first line handles the base case, where the first item in the list is the one we want to delete
- The second line handles the case where the item we want to delete ISN'T the first item
- Our rule uses pattern matching to break up the input list into the Head item and all Tail items.
- the subgoal: "Use delete to remove the Item from amongst the Tail items; FinalTail refers to the resulting tail"
- In this case, we construct our output list by concatenating the Head item from our original list with the tail of the list with the Item removed from it.

Prolog Lists: Built-in Facts and Rules

`append(X,Y,Z)` - determines if list X concatenated with list Y is equal to list Z

```
append([1,2],[3,4],[1,2,3,4]) yields True
append([1,2],X,[1,2,3,4]) yields X -> [3,4]
```

`sort(X,Y)` - determines if the elements in Y are the same elements of X, but in sorted order

```
sort([4,3,1], [1,3,4]) yields True
sort([4,3,1],X) yields X -> [1,3,4]
```

`permutation(X,Y)` - determines if the elements in Y are the same elements of X, but in a different ordering

```
permutation([4,3,1], [3,1,4]) yields True
```

`reverse(X,Y)` - determines if list X is the reverse of list Y

```
reverse([1,2,3],[3,2,1]) yields True
reverse([1,2,3],X) yields X -> [3,2,1]
```

`member(X,Y)` - determines if X is a member of the list Y

```
member(6, [1,6,4]) yields True
member(X,[1,6,4]) yields X -> 1, X -> 6, and X -> 4
```

false: member([ ],[ ]).

true: member([ ], [_]). member([_], [_]). member([ ], _). member([_], _). member(_,[_]). member(_,_).

`sum_list(X,Y)` - determines if the sum of all elements in X add up to Y

```
sum_list([4,3,1], 8) yields True
sum_list([4,3,1], Q) yields Q -> 8
```

List Syntax is Syntactic Sugar For Functors and Atoms!

- list processing is implemented just like any other Prolog fact or rule!
- the empty list [] can similarly be written as the atom nil
- The `|` operator can be replaced with a fact named "cons" that takes two arguments

```
[X|Tail] --> cons(X,Tail)
```

Examples

insert lexicographically

```
insert_lex(X,[],[X]).
insert_lex(X,[Y|T],[X,Y|T]) :- X =< Y.
insert_lex(X,[Y|T],[Y|NT]) :- X > Y, insert_lex(X,T,NT).
```

```
equal_lists([],[]).
equal_lists([HeadA|TailA], [HeadB|TailB]) :-
    HeadA = HeadB,
    equal_lists(TailA, TailB).
```

```
is_palindrome(List) :-
    reverse(List, Rev),
    equal_lists(List, Rev).
```

```
make_palindrome([],[]).
make_palindrome(List, Pal) :-
    reverse(List, [_|ReverseTail]),
    append(List, ReverseTail, Pal)
```

```
reverseList(L1, L2) :- reverseHelper(L1, L2, []).

reverseHelper([], L2, L2).
reverseHelper([X|T], L2, Acc) :- reverseHelper(T, L2, [X|Acc]).
```